Configuring Oracle Server for VLDB

Cary V. Millsap
Oracle Corporation

August 21, 1996

Abstract

This paper will help the reader configure a very large Oracle Server database (VLDB) for high performance and high availability with low maintenance. It describes decisions about database block size, RAID technology and raw devices, redo log files, tablespace partitioning, storage parameters, and rollback segments. This paper explores the technologies and trade-off constraints associated with these issues and presents technically detailed methods for optimizing a configuration within these trade-offs.

Contents

1. Introduction

VLDB is an acronym for *very large database*. What constitutes “very large” means different things to different people, but the label always conjures an impression of difficulty and complexity, cost and risk. *VLDB* is what people call the most formidable databases that state of the art technology will support. It takes a lot of creativity, smart planning, investment, and hard work to prevent an attempted VLDB implementation from becoming a very large disappointment.

1.1 Operational Complexity

Operational complexity is an obvious VLDB challenge. As the operator of a VLDB, you are continually evaluating dozens of tightly interdependent parameters. The system will penalize attempts to make drastic or arbitrary moves. Big objects and time-consuming processes leave you little maneuverability to get out of trouble, so you have to plan carefully to avoid the trouble in the first place. The livelihood of several hundreds or even thousands of people depend upon your ability to make good decisions quickly. It takes talent and hard work to do these things well.

Many essential operational procedures like backup and restore consume time proportional to the size of the objects being manipulated. So for a very large database, these essential operations take a very long time. Mistakes become more costly as a database grows. Repair operations—like reconfiguring a disk, rebuilding a table, counting the rows in a table, undoing a transaction—might seem normal in a smaller database, but they cannot be tolerated in a VLDB. If an activity takes hours out of your operational uptime, you do what you must to avoid it.

1.2 High Performance

Other factors compound the difficulty. Consider data access. You may be able to tolerate an inefficiently written query that consumes 2 CPU seconds on a high-powered modern system with 20 users, but you

cannot tolerate the same 2-second service time on a system with 1,200 users all fighting for latches, disks, and CPU cycles. Queries against VLDB data must be exceptionally well tuned, or the whole world falls apart.

There's more. Many VLDBs become "VL" in the first place because large numbers of simultaneous users and batch programs induce high transaction rates. Very high transaction rates put a lot of stress on the I/O subsystems associated with redo generation and database writing. And very high process concurrency puts a lot of stress on the latches and locks designed to serialize access to critical resources.

1.3 High Availability

The more you think about it, the harder it seems. VLDBs are often the data sources driving mission critical applications with very high availability requirements. High-end database designers hear "seven-by-twenty-four" until we're nearly numb to the idea of how monumentally difficult it is—in the face of electrical and network failures, bad memory boards and disk controllers, application and operating system bugs, software and hardware upgrades, and occasionally even an imperfect operator—to achieve only seconds or minutes of system downtime each year. How does one detect and repair the logical corruption of hundreds or even thousands of gigabytes of data that can be caused by an operator entry error?

1.4 Overcoming VLDB Challenges

The way to overcome these challenges is to make your database seem as small as possible. You optimize both your configuration and your application to reduce load at every opportunity. You choose data structures that minimize query access I/O. You create applications with the lightest transactions that can possibly meet the application's functional requirements. You isolate the impact of component failures to the smallest possible subset of your data. You make your backup and recovery units as small as possible.

There are several aspects to doing VLDB well, including:

- Optimizing the functional process flow
- Optimizing the logical data model
- Optimizing the schema
- Constructing a reliable, high performance hardware configuration
- *Optimizing the physical database configuration*
- Optimizing the application SQL
- Optimizing the operating system and Oracle Server instance
- Creating correct, reliable operational procedures

In this paper, we investigate the physical database configuration decisions required to build a successful VLDB.

2. Oracle Block Size

One of life's ironies is that on the day when your Oracle Server experience is the least it will ever be, you are required to type a value for the **db_block_size** database parameter which will follow that database throughout its lifetime. It happens that the value you choose for your Oracle database's block size is of profound importance to the performance of your system. The following sections will alert you to some of the trade-offs you should consider before choosing the Oracle block size for your database.

Optimal Oracle block sizes for VLDB range from 4KB to 64KB. The most common VLDB block size is probably 8KB, with 16KB in second place. Oracle Server implementations on systems capable of very fast memory transfer of large memory objects can benefit from 32KB Oracle blocks and (theoretically, at least) 64KB. Under no circumstances of which I am aware should a VLDB operate with a 2KB block size, and there are only a few very special cases in which a 4KB block size would be appropriate for a VLDB. In [Millsap (1996)], I present a much deeper technical description of the issues we shall summarize here.[1]

2.1 Reducing I/O Load

The most important characteristic of the most difficult VLDB implementations in the world is the enormous I/O load on these systems. The owners of the best VLDBs in the world invest extensively into any technique possible for reducing the I/O load on their systems. Using big Oracle database blocks is

[1] Rather than tempt you to find an Oracle Corporation internal document that won't be easy to find, the special case for which 4KB blocks may be appropriate is the VLDB that consists of a huge number (thousands) of very small segments (smaller than 100KB each).

one simple technique for reducing some of this I/O load.

- *I/O call overhead*—The hardware fixed overhead costs of any single I/O call dominate the total time for fulfillment of the call. Manipulating fewer bigger Oracle blocks thus yields a performance advantage over manipulating more smaller blocks.
- *Index performance*—Index performance is inversely proportional to the height of the B*-tree index structure. Larger Oracle blocks allow for storage of more keys per block, which yields shallower, wider indexes with better search performance.
- *Reduced chaining*—Oracle Server uses *chaining* to store table, data cluster, or hash cluster rows that will not fit into a single database block. Chaining of row pieces across multiple blocks results in multiple I/O calls per manipulated row. Bigger Oracle blocks reduce the probability that a row will require chaining, hence reducing the total number of I/O calls in the system.

2.2 Improving Space Efficiency

Oracle Server stores a small amount of overhead in each Oracle block whose size is constant, irrespective of the amount of data stored within the block. Thus, the fewer blocks stored in a database, the less total fixed overhead that database will consume. For very small segments, the fixed overhead savings can be overwhelmed by wasted space present in the last few unused blocks of a segment, but this waste is itself generally overwhelmed by imprecise storage parameters.[2] For very large segments, using a larger Oracle block size saves roughly two to six percent in total database volume. This savings translates to valuable performance and operational cost benefits, especially noticeable in backup/restore times.

2.3 Preventing Concurrency Bottlenecks

Using bigger Oracle blocks increases the risk of concurrency bottlenecks unless the physical database architect understands how to use the **initrans** and **maxtrans** storage parameters. The value of **initrans** should be set to the maximum number of transactions that will be expected to simultaneously transact within the block.[3] Hence, if you were to rebuild your database with *k* times larger Oracle blocks, then you should increase your **initrans** values by a factor of *k* as well. To enable dynamic growth of the "transaction entry table list," you must also set **maxtrans** to a value larger than **initrans**.[4]

Failing to choose proper values of **initrans** or **maxtrans** for your database segments can cause waits in user sessions. Sessions that update Oracle blocks with this problem will prevent other sessions from obtaining entries in the block's data structure that enables reconstruction of read-consistent queries in that block. The so-called *ITL contention* that results can be detected by database administrators who monitor **v$lock**. They will see a session waiting for a **TX** (transaction enqueue) lock in lock mode **4** (share).

You can completely avoid these problems by using sound techniques described in the standard Oracle Server documentation for setting **initrans** and **maxtrans** for database segments.

2.4 Trade-Offs

Although larger-than-normal Oracle block sizes are good for VLDBs, there are limits to the maximum Oracle block size you should choose.

- *Physical I/O size*—The size of an Oracle database block should not exceed the maximum size of a physical read. Be aware also that the size of an operating system physical read also constrains the actual size of batched reads on full-table scans. For example, if your Oracle block size is 32KB and your multiblock read count is 32, you very likely will not be fetching $1MB = 32 \times 32KB$ per I/O call.
- *Redo size*—The Oracle Server redo log writer (LGWR) writes entire Oracle blocks to the on-line redo log files for transactions that are executed on tablespaces that are in hot backup mode. Hence, your desire to reduce your system's redo volume during hot backups may motivate you to limit your Oracle database block size.
- *Memory copy size*—If you choose your Oracle block size to be larger than the amount of memory that your operating system or hardware architecture will support in one operation, you

[2] As we shall discuss later, having storage parameter imprecision is actually a good trade-off compared to the operational costs of administering very precise storage parameters in most VLDBs.

[3] *Transact* here means "performs an **insert**, **update**, **delete**, or **select…for update**."

[4] This "transaction entry table" is formally called the *interested transactions list*, or *ITL*.

may note higher CPU loads and performance degradation associated with manipulating large blocks in the SGA.

- *Concurrency limitations*—The maximum allowable value for **initrans** and **maxtrans** is 255. If your Oracle block size gets so large that more than 255 transactions will routinely attempt to update a single Oracle block at the same time, then your block size is too large. The chances of this occurring to you are slim to none: if you actually tried to put 255 concurrent transactions on one block, the resulting latch contention would be so severe that you'd have to increase **pctfree** for the segment.

3. Disk Configuration

A good database server is at its heart a reliable, high-performance I/O manager that must operate within its owner's economic constraints. Understanding this makes it easy to understand why I/O subsystem configuration is such an important topic for the architect of any VLDB application.

3.1 Structured Configuration Analysis

At the highest level of abstraction, defining system requirements is easy, because everyone has the same three desires: *fast*, *reliable*, and *inexpensive*. However, when we convert these general philosophical goals into something real, we are confronted with trade-off decisions. To navigate these conflicts, you must do two things well: (1) know your goals, and (2) know your technology. The trick is to find the right *mixture* of fast, reliable, and inexpensive to meet your specific business needs. It seems that the right mixture is never the same for you as it is for your neighbor.

Architectural constraints come at us from two directions: (1) there are economic constraints, and (2) there are technology constraints. Economic constraints you already understand very well, and I'm sure you wish you didn't have to deal with them. However, even in an imaginary universe in which you're permitted to buy as much hardware and software as you could possibly want, you'd still face technology constraints.

For example, let's say that you have an OLTP application, and you have optimized your disk configuration for Oracle Server data files to give uncompromised random I/O performance at the expense of some sequential I/O performance degradation. This may seem like a low-impact trade-off for OLTP because Oracle Server OLTP I/O to data files consists predominantly of massive numbers of well behaved indexed or hash-based reads, and massive numbers of scattered DBWR writes.

However, your first index rebuild event on a disk array containing data accessed at a high concurrency level may well consume 300–500 percent more time than if you had used a different disk configuration. Your decision intended to optimize on-line performance without compromise has directly reduced your system's availability by extending the duration of index unavailability during the index rebuild.

There are hundreds of lessons like this one that make us appreciate the importance and difficulty of knowing your requirements and knowing your technology. The most important tool you can have is a structured analysis method that helps you ask all the right questions. In this paper, we will structure our analysis of I/O subsystem construction by decomposing *performance*, *availability*, and *cost* as follows:

Performance
- random read performance
- random write performance
- sequential read performance
- sequential write performance
- impact of concurrency

Availability
- outage frequency
- outage duration
- performance degradation during outage

Cost
- acquisition cost
- operational cost

In this paper, we evaluate several tools and techniques in the context of these categories, which are described in the following sections.

3.1.1 Performance

- *Random read performance*—Examples include indexed or hash-based queries, and rollback segment reads. Presumably this category includes the bulk of the read calls executed on a high-volume OLTP system. For a data warehouse, this category may represent only a small fraction of total load.
- *Random write performance*—Examples include all DBWR writes of data, index, and undo (rollback) segments. This category accounts for a significant proportion of total write calls on an

OLTP system during normal operation. There may be little or no regular random write activity on a data warehouse.

- *Sequential read performance*—Examples include backups, full-table scans, index creations, parallel queries, temporary segment reads, and recovery and roll-forward from archived redo log files. Even for well optimized OLTP applications, this category still accounts for a considerable proportion of total load, especially on OLTP systems with heavy batch processes. On a data warehouse, sequential reads may account for nearly all of the system's read calls.
- *Sequential write performance*—Examples include LGWR writes, temporary segment writes, direct-path data loads, index creations, tablespace creations, and data restore operations. Designers who are focused on transaction and query processing sometimes forget this category. But initial data uploads and operational maintenance in any type of Oracle database generate a lot of sequential-write load.
- *Impact of concurrency*—In each of the categories named above, the architect should consider the impact of varying concurrency levels.

3.1.2 Availability

- *Outage frequency*—Outage frequency is the expected number of occurrences of a possible outage per unit of time. Outage frequency is specified with the MTTF (mean time to failure) metric. For example, a disk drive with an MTTF of 200,000 hours can be expected to fail only once every 23 years.
- *Outage duration*—Outage cost may be expressed in dollars, which in turn can be expressed as a function of outage duration. Outage cost for a given event is proportional to the MTTR (mean time to repair) metric for that event—the longer the outage duration, the higher the outage cost.
- *Performance degradation during outage*—Different disk configurations yield different performance levels during outage events. Some configurations have no fault tolerance whatsoever; some configurations provide service during fault at degraded performance levels; and some configurations provide performance at full strength during different types of outage. It is important to match your hardware's outage degradation levels with your application's requirements.

3.1.3 Cost

- *Acquisition cost*—Acquisition cost is the economic cost of purchasing the system.
- *Operational cost*—Operational cost is the cost of running and maintaining the system. Operational costs include both maintenance costs that depend on the reliability of the system and administrative costs that depend on the complexity of the system. For example, a system that requires less easily obtainable (e.g., smarter or better trained) operators and repair technicians will cost more to operate.

3.1.4 Analysis Method

Decomposing performance, reliability, and cost this way gives us important tools to design a good system. It allows us to evaluate the trade-offs both within a category (e.g., random read performance *vs.* sequential write performance) and understand one category in terms of another (e.g., the strong link between sequential write performance and outage duration).

These categories also formalize the way we think about our system. For example, understanding the decomposition of reliability allows us to view the dimensions of outage independently: (1) we can focus on reducing the frequency of an outage whose cost we know we cannot tolerate, or (2) we can focus our efforts on reducing the duration of an outage we know we cannot prevent. And it reminds us that outage duration is partly a reliability issue and partly a performance issue.

The following sections will help guide you through a high-level analysis of the performance, availability, and cost trade-offs in Oracle Server I/O subsystem design. The aim of this section is to help you better understand your goals and technology options to help you make more informed decisions, so you can get the best results from your investment.

3.2 RAID

In second place behind the human being, the most unreliable mission-critical component of a computing system is the disk drive. To improve disk reliability, most hardware vendors today market disk array systems called *RAID*—redundant arrays of inexpensive disks. RAID offers high-performance, fault-resilient I/O subsystems using less expensive

disk drive technology than that found on traditional high-availability mainframe systems.

The term *RAID* was created in a 1987 paper published by Patterson, Gibson, and Katz at the University of California [Patterson et al. (1988)]. The numeric levels of RAID represent different organizations of common minicomputer disk technology to accomplish performance and reliability goals at low cost.

The primary RAID levels that the Oracle VLDB architect should understand are levels 0, 1, 0+1, 3, and 5. Figure 1 summarizes the principal concepts of these RAID configurations. Note that hardware vendors can choose to implement RAID's striping and mirroring functions either in hardware or in software. These choices affect the number and types of special controllers required for implementation.

The performance advantages of RAID configurations are stunning. By distributing physical I/Os uniformly across an array of disk drives, striped RAID configurations provide unmatched response time and throughput statistics. A five-disk striped RAID 0 array can serve almost nine times as many random I/Os as a configuration of five independent disks with equivalent response times. The same five-disk striped array can provide almost five times the sequential I/O throughput of an independent disk configuration of equal size.

The reliability advantages of RAID are equally impressive. A disk system using one hundred 200,000-hour MTTF disk drives configured in a non-redundant organization has a system mean time to data loss (MTTDL) of less than 83 days. The same 100-disk system configured in a redundant RAID organization has an MTTDL of about 26 years [Chen et al. (1992), 161–163]. Mean-time-to-recovery advantages of RAID are excellent as well. You can pull an active drive out of a RAID level 5 disk tray, and the system will continue without missing a beat.

However, each RAID configuration bears unique costs. You may find that the RAID configuration you really need is too expensive, and that the only configurations you can afford impose too many other compromises. The Oracle system architect must navigate a twisty maze of complex trade-offs when deciding how to use RAID.

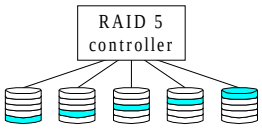
Several authors do a brilliant job of describing RAID configurations and evaluating RAID organizations on the basis of reliability, performance, and cost [Chen et al. (1994); Gui (1993); Sun (1995); and many others]. The following sections contain summaries of these concepts with concrete recommendations about when and how to use each type of RAID configuration for Oracle Server applications.

3.2.1 Definitions

Studying RAID literature can be confusing because the authors in the market seem to interpret the definitions however they like. To make this paper as simple as we can, let's define some important terms before going on.

- *Array*—RAID configurations at levels 0, 0+1, 3, and 5 group disks into collections called *error correction groups* or *disk arrays*. We shall call a group of disks that make up an error correction group an *array*.
- *Stripe*—S*triping* is a hardware or software capability in which logically contiguous data are written in pieces that are distributed across the disks in an array.[5] These pieces are called *striping segments* or *stripes*.

In reading this document, it will be important to remember that an *array* is a collection of disks, and that *stripes* are the pieces that are distributed across an array. The following picture shows one disk array containing five disks. Each disk contains five stripes, for a total of 25 stripes in the entire picture.



[5] Note that *striping* is spelled with one *p*, as opposed to *stripping* with two *p*'s, which is what one does to old paint.

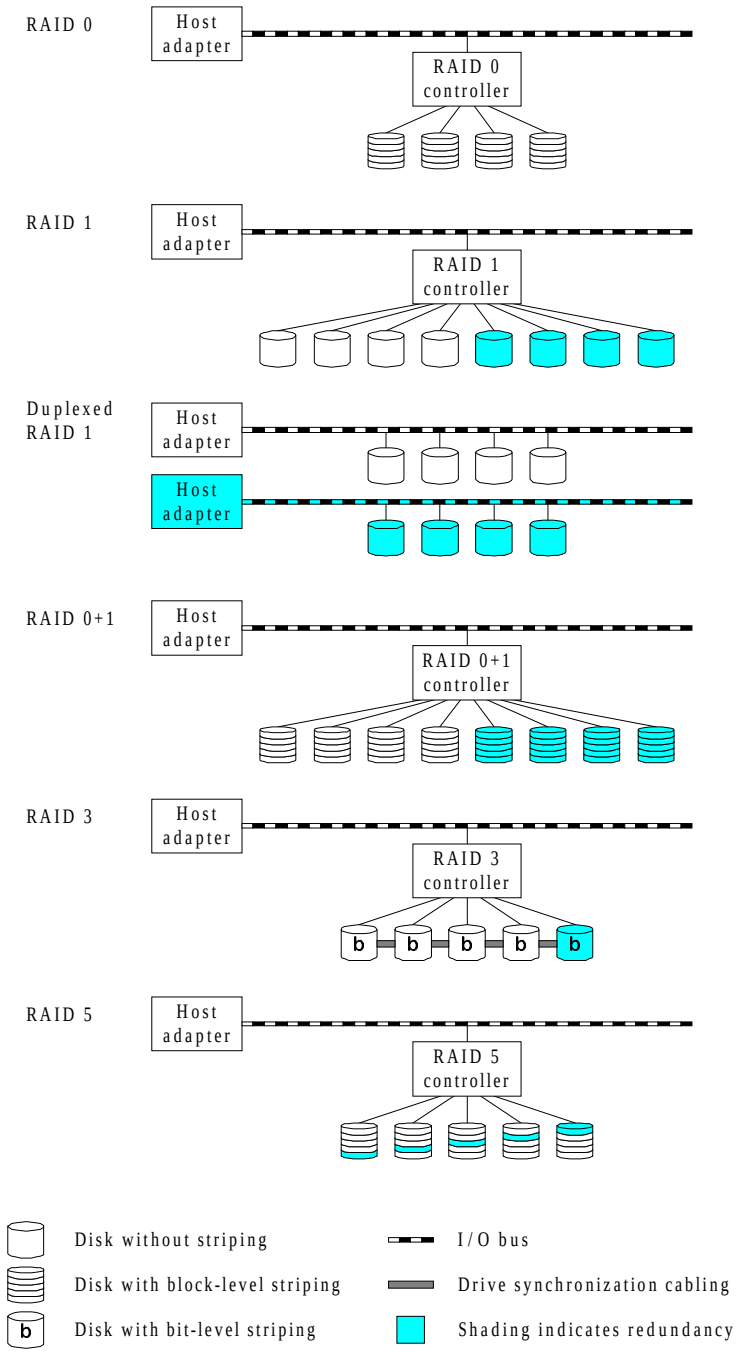


Figure 1. Overview of RAID hardware architectures.

3.2.2 Nonredundant Striping (RAID Level 0)

RAID 0 is a nonredundant disk configuration that may be striped. Properly configured striping yields excellent I/O response times for high concurrency random I/O and excellent throughput for low concurrency sequential I/O. Selection of stripe size for an array requires careful consideration of trade-off constraints. We shall discuss the details of stripe size optimization in a later section.

Nonredundant disk is useful only for environments in which capital acquisition cost constraints outweigh system reliability requirements.

- *Random read performance*—Excellent under all concurrency levels if each I/O request fits within a single striping segment. Using a stripe size that is too small can cause dramatic performance break-down at high concurrency levels.
- *Random write performance*—Same as random read performance.
- *Sequential read performance*—Excellent with fine-grained striping at low concurrency levels. Also excellent at high concurrency levels if each I/O fits within a single striping segment. Using stripe sizes that are too small causes dramatic performance break-down at high concurrency levels.
- *Sequential write performance*—Same as sequential read performance.
- *Outage frequency*—Poor. Any disk failure will cause application outage requiring Oracle media recovery for every application with data stored in the failed disk's array.
- *Outage duration*—Poor. The duration of a RAID 0 outage is the time required to detect the failure, replace the disk drive, and perform Oracle Server media recovery.
- *Performance degradation during outage*—Poor. Any disk failure incurs total blackout for all applications requiring use of the array containing the failed disk until Oracle media recovery is complete.
- *Acquisition cost*—Excellent. RAID 0 is the least expensive of all RAID configurations to implement.
- *Operational cost*—Fair to poor. Operational costs of dealing with frequent media recoveries may outweigh the acquisition cost advantages of RAID 0 over redundant configurations. Increasing capacity requires either purchase of one or more entire arrays, or reconfiguration of the existing arrays. Training is required to configure striped disk arrays for optimal performance.

3.2.3 Mirroring (RAID Level 1)

Mirroring is the VLDB architect's fundamental tool for reducing the frequency of disk outage. Mirrored disks are systems in which identical copies of your data are written to two or more disks on every write call, enabling your application to keep running as long as at least one image is left undisturbed.

In spite of some write performance degradation relative of mirroring relative to nonredundant configurations, mirrored disks yield the best write performance of the fault resilient disk configurations. Mirroring is especially valuable for Oracle data files holding files with high write rates. Many Oracle architects use mirroring to protect on-line and archived redo log files even when cost constraints prohibit mirroring the entire disk subsystem.

Multiplexing host adapters, busses, and power supplies makes a mirrored system even more impervious to hardware failure. In a multiplexed mirrored configuration, *n*-way redundant writes are performed in software to *n* identical host adapters. Several Oracle customers today use three-way multiplexed ("triple-mirrored") configurations to provide flexibility for data restore operations. Triple mirroring provides the normal fault resilience of duplexed mirroring and allows a database administrator to restore the database to a past point in time without using tapes. We shall discuss one triple mirroring technique in a later section.

- *Random read performance*—Good. If the implementation uses read-optimizing RAID 1 controllers, then marginally better than an independent disk; otherwise, identical to an independent disk. An optimizing RAID 1 controller will service a read request by reading only the image whose drive requires the smallest I/O setup cost, leaving the other disk(s) in the mirror group free to service other read requests in parallel.
- *Random write performance*—Good. If the implementation uses read-optimizing RAID 1 controllers, then marginally worse than an independent disk; otherwise, identical to an independent disk. Although the mirrored writes can execute in parallel, the speed of a write call is constrained by the speed of the slowest of the

mirrored writes. The cost of an *n*-way RAID 1 write is $\max(w_1, w_2, \ldots, w_n)$, where w_i is the cost of writing to the *i*th mirror piece. Read-optimizing controllers desynchronize mirrored disk drive states, making $w_i \neq w_j$ for $i \neq j$.

- *Sequential read performance*—Fair. See random read performance. Throughput is limited to the speed of one disk.
- *Sequential write performance*—Fair. See random write performance. Throughput is limited to the speed of one disk.
- *Outage frequency*—Excellent. An *n*-way mirrored disk system can withstand up to $n-1$ disk failures per mirror set without incurring application outage. However, RAID 1 does not eliminate the host adapter, I/O bus, RAID controller, power supply, firmware bugs, or software bugs as single points of failure. Multiplexing eliminates the hardware failure points. An *n*-way multiplexed disk system can withstand up to $n-1$ host adapter, I/O bus, or power supply failures per plex, *and* up to $n-1$ disk failures per mirror set without incurring application outage. Multiplexed mirrored disk is the most fault-resilient configuration available.
- *Outage duration*—Excellent. The duration of a partial outage involving $n-1$ or fewer drives in one or more mirror groups or $n-1$ components in a multiplex is the amount of time required to replace the failed component. During this time there will be no application outage. The duration of a data loss outage involving *n* drives in a mirror group or *n* components in a plex is the same as the duration of a RAID 0 outage. During this time, there will be full application blackout.
- *Performance degradation during outage*—Excellent. There is no performance degradation incurred by taking a RAID 1 disk off-line. However, upon replacement of a failed drive, the resilvering operation for the fresh drive will compete with all other I/O to the repaired mirror group, causing significant performance degradation for applications reading or writing the repaired mirror group for the duration of the reconstruction. On an *n*-way multiplexed implementation, up to $n-1$ host adapter, I/O bus, or power supply failures incur no performance degradation during outage.
- *Acquisition cost*—Poor. Capacity cost for *n*-way mirroring is *n* times that of equivalent RAID 0 capacity. RAID 1 implementations require special RAID 1 controllers in addition to the same number of host adapters required by RAID 0 implementations. *n*-way multiplexed RAID 1 requires *n* times as many host adapters as RAID 1 but no special RAID 1 controllers. Total multiplexed disk capacity is constrained by I/O subsystem host adapter bay capacity because *n*-way multiplexing consumes *n* times more adapter slots than non-multiplexed configurations.
- *Operational cost*—Fair. Costs include training of staff in effective administration of the mirrored system, costs to develop custom software to integrate mirroring procedures into scheduled maintenance operations, and so on.

3.2.4 Striping and Mirroring (RAID Level 0+1)

Striping and mirroring can be combined to form what many people today call "RAID 0+1."[6] RAID 0+1 unites all of the advantages and disadvantages of RAID levels 0 and 1: excellent performance and excellent fault resilience with high acquisition cost. Duplexed mirroring of striped disk arrays is the superior configuration for mission critical, high volume OLTP applications.

- *Random read performance*—Excellent under all concurrency levels if each I/O request fits within a single striping segment. Using a stripe size that is too small can cause dramatic performance break-down at high concurrency levels. Marginally better than RAID 0 if using RAID 1 read optimization.
- *Random write performance*—Excellent, with excellent high-concurrency performance. Marginally worse than RAID 0, but much better than RAID 5. See section on RAID 1 random write performance.
- *Sequential read performance*— Excellent under all concurrency levels if each I/O request fits within a single striping segment. Using a stripe size that is too small can cause dramatic performance break-down at high concurrency levels. Marginally better than RAID 0 if using RAID 1 read optimization.

[6] Note that the inventors of RAID terminology include the provision for "RAID 0+1" within the term RAID 1 itself [Chen et al. (1994), 153]. Hardware vendors appear to have guaranteed the term "RAID 0+1" a rich and happy life.

- *Sequential write performance*—Excellent. Marginally worse than RAID 0, but better than RAID 5. See section on RAID 1 random write performance. Using a stripe size that is too small can cause dramatic performance breakdown at high concurrency levels.
- *Outage frequency*—Excellent. Same as RAID 1.
- *Outage duration*—Excellent. Same as RAID 1.
- *Performance degradation during outage*—Excellent. Same as RAID 1.
- *Acquisition cost*—Poor. Same as RAID 1 with potential added cost for striping hardware or software.
- *Operational cost*—Fair. Optimizing striped configurations requires training, as does integrating mirror operations into maintenance procedures. Increasing disk capacity requires addition of entire arrays or reconfiguration of existing arrays.

3.2.5 Bit-Interleaved Parity (RAID Level 3)

RAID 3 is one answer to the high cost of 1-for-1 redundancy of RAID 1. In a RAID 3 configuration, disks are organized into arrays in which one disk is dedicated to storage of parity data for the other drives in the array. The RAID 3 striping unit is 1 bit. The configuration has the property that any single data drive's data can be reconstructed given the data of the remaining drives on its array using an exclusive-or operation.

The primary advantage of RAID 3 is that its acquisition cost is much less than that of RAID 1. However, its poor random I/O performance and its poor performance during partial outage make RAID 3 generally unsuitable for most Oracle Server applications. RAID 3 best suits an Oracle application whose economic constraints outweigh its reliability requirements, whose reads are mostly poorly optimized full-table scans, and which performs virtually no inserts, updates, or deletes.

- *Random read performance*—Poor. Drive synchronization prohibits parallelization of small random read calls. Poor high-concurrency performance.
- *Random write performance*—Poor. Drive synchronization also prohibits parallelization of small random write calls. Poor high-concurrency performance.
- *Sequential read performance*—Very good for low concurrency applications; worse for higher concurrency applications.
- *Sequential write performance*—Very good for low concurrency applications; worse for higher concurrency applications.
- *Outage frequency*—Good. A RAID 3 system can withstand the loss of any single disk on a given array without incurring application outage. Loss of two disks on an array incurs outage that must be repaired by media recovery. Note that the resilience of a RAID 3 implementation degrades as the number of disks per array increases, and that this resilience degradation in fact occurs at a higher rate than the acquisition savings noted above. For example, doubling the number of drives in a RAID 3 array from 5 to 10 will save approximately 14% in acquisition costs (see section on RAID 3 acquisition cost), yet doubling the array size increases the outage frequency of an array by 100%.
- *Outage duration*—Good. Duration of partial outage caused by loss of one disk in an array is the amount of time required to detect the failure and replace the drive. Duration of full outage caused by loss of two or more disks in an array, a host adapter, bus, or other nonredundant component includes the time required to perform Oracle Server media recovery.
- *Performance degradation during outage*—Fair. Loss of a dedicated parity drive incurs no performance penalty upon the application until the drive is replaced. Loss of a data drive incurs a significant performance penalty upon the application before the drive is replaced because every I/O to the array with the failed drive will require a calculation in turn requiring I/O on every other drive in the array. These I/Os cannot be parallelized within an array. Upon replacement of any failed RAID 3 drive, the I/O required to reconstruct the data on the replacement drive will compete with all other I/O to the array, causing performance degradation.
- *Acquisition cost*—Fair. Disk capacity cost is $g/(g-1)$ times the cost of equivalent RAID 0 capacity, where *g* is the number of disks in an array, plus the additional cost of special drive-synchronized disks and special controllers required for RAID 3. Thus, the acquisition cost of RAID 3 is always greater than the cost of

RAID 0, but generally less than RAID 1 for $g > 2$. Note that total capacity cost is reduced by increasing the number of disks per array. For example, using five disks per array results in a disk capacity cost of 125% times the cost of equivalent RAID 0 capacity, but using ten disks per array results in a disk capacity cost of only 111% the cost of RAID 0. RAID 3 implementations also require special RAID 3 controllers in addition to the host adapters required by RAID 0.

- *Operational cost*—Fair. Training required to create operational procedures dealing with different outage events. Capacity growth requires purchase of entire arrays or reconfiguration of existing arrays.

3.2.6 Block-Interleaved with Distributed Parity (RAID Level 5)

RAID 5 is similar to RAID 3 except that RAID 5 striping segment sizes are configurable, and RAID 5 distributes parity across all the disks in an array. A RAID 5 striping segment contains either data or parity. Any write to a RAID 5 stripe requires resource intensive six-step process [Sun (1995)]:

1. Read the blocks to be overwritten.
2. Read the corresponding parity blocks.
3. Remove the contribution of the data to be overwritten from the parity data.
4. Add the contribution to parity of the new data.
5. Write the new parity data.
6. Write the new data.

Battery-backed cache greatly reduces the impact of this overhead for write calls, but its effectiveness is implementation dependent. Large write-intensive batch jobs generally fill the cache quickly, reducing its ability to offset the write-performance penalty inherent in the RAID 5 definition.

RAID 5 is useful for reducing the cost of disk subsystems for any data requiring good fault resilience with high read rates, but not for data requiring high write rates. Because of RAID 5's aggressively poor write performance, the attitude of many experienced VLDB architects toward RAID 5 is *just say no*. For read-intensive VLDBs in which the media recovery time penalty of poor write performance is not as important as acquisition cost constraints, RAID 5 offers acceptable performance at much lower cost than RAID 1.

- *Random read performance*—Excellent under all concurrency levels if each I/O request fits within a single striping segment. Using a stripe size that is too small can cause dramatic performance break-down at high concurrency levels.
- *Random write performance*—Poor. Worst at high concurrency levels. The read-modify-write cycle requirement of RAID 5's parity implementation penalizes random writes by as much as an order of magnitude compared to RAID 0. Disk cache helps performance if the cache is large enough to handle the concurrency level.
- *Sequential read performance*—Excellent with fine-grained striping at low concurrency levels. Also excellent at high concurrency levels if each I/O fits within a single striping segment. Using stripe sizes that are too small causes dramatic performance break-down at high concurrency levels.
- *Sequential write performance*—Fair for low concurrency levels. Up to an order of magnitude worse than RAID 0 for high concurrency levels. Large writes tend to fill the RAID 5 disk cache, reducing its ability to moderate poor write performance. As with sequential reads, higher concurrency levels degrade fine-grained striping performance.
- *Outage frequency*—Good. A RAID 5 system can withstand the loss of any single disk on a given array without incurring application outage. Loss of two disks on one array incurs outage that must be repaired by media recovery. Note that the resilience of a RAID 5 implementation degrades as the number of disks per array increases, and that this resilience degradation occurs at a higher rate than the acquisition savings. See RAID 3 *outage frequency* for details.
- *Outage duration*—Good. Duration of partial outage caused by loss of one disk in an array is the amount of time required to detect the failure and replace the drive. Duration of full outage caused by loss of two or more disks in an array, a host adapter, bus, or other nonredundant component includes the time required to perform Oracle Server media recovery.
- *Performance degradation during outage*—Fair. There is no degradation for reads from a surviving drive in array containing a failed drive. Writes to a surviving drive require a read-modify-write performance penalty. Reads and

	RAID Level					
	None	0	1	0+1	3	5
Control file performance	2	1	2	1	5	3
Redo log file performance	4	1	5	1	2	3
system tablespace performance	2	1	2	1	5	3
Sort segment performance	4	1	5	1	2	3
Rollback segment performance	2	1	2	1	5	5
Indexed read-only data files	2	1	2	1	5	1
Sequential read-only data files	4	1	5	1	2	3
DBWR-intensive data files	1	1	2	1	5	5
Direct load-intensive data files	4	1	5	1	2	3
Data protection	4	5	1	1	2	2
Acquisition and operating costs	1	1	5	5	3	3

Figure 2. Estimated relative rankings for RAID configurations from 1 (best) to 5 (worst) for specific Oracle file types. RAID 0+1 is the superior technical configuration option, and it is also the configuration with the highest cost. Judicious use of RAID 5 arrays allows the disk subsystem architect to reduce system cost at minimized performance and availability sacrifice. Adapted from [Sun (1995), 28].

writes to a failed drive incur a high performance penalty, requiring data from all the surviving drives in the failed drive's array. Reconstruction of a replaced drive dramatically degrades performance to the array by competing with other I/O requests to the array.

- *Acquisition cost*—Fair. Disk capacity cost is $g/(g-1)$ times the cost of equivalent RAID 0 capacity, where g is the number of disks in an array, plus the additional cost of RAID 5 controllers. RAID 5 acquisition cost is always greater than the cost of RAID 0 but generally less than RAID 3 and theoretically less than RAID 1 for $g > 2$. In real-life, performance expectations of RAID 5 implementations sometimes exceed the configuration's ability to deliver. The resulting analyses and equipment acquisitions sometimes drive the cost of RAID 5 above that of RAID 0+1.
- *Operational cost*—Fair. Training is required to configure striped disk arrays for optimal performance. Capacity growth requires purchase of entire disk arrays or reconfiguration of existing arrays.

3.2.7 RAID Summary

Disk technology is improving so quickly that it is difficult to summarize the performance of several different implementations of several technologies without making some sweeping statements that are bound to be untrue in specific circumstances. If you live by the golden rule of *know your goals, know your technology*, then you will periodically reevaluate your choices. The best way to know your goals and know your technology is to learn as much as you can from other people's experiences and test your theories thoroughly. As your business risk rises, so too rises the importance of thorough testing. Figure 2 shows the estimated relative merit of various RAID configurations specifically for Oracle Server implementations.

3.3 Stripe Size

Striping will benefit you only if you optimize your choice of your stripe size. Because different stripe sizes optimize different operations, the best VLDB configuration will generally require use of different stripe sizes on different disk arrays in a system. Striping yields its most popularly heralded advantages when the stripe size is small. However, using fine-grained striping for the wrong operations can be

disastrous. In the following sections, we shall explore how to exploit fine-grained striping while avoiding its disadvantages.

3.3.1 Fine-Grained Striping

Fine-grained striping configurations interleave data in small units across a disk array so that all I/O requests, regardless of size, access all of the disks in the array. The advantage is a very high data transfer rate for all I/O requests. The disadvantages of fine-grained striping are [Chen et al. (1993), 151]:

- Only one logical I/O request can be in service at any given time on an array. That is, parallelization of a single request comes at the expense of not being able to execute larger numbers of requests in parallel.
- All disks must waste time positioning for every request. That is, if the stripe size is smaller than an I/O request, then two or more drives must endure the cost of an I/O setup for the request.

We can avoid these disadvantages of fine-grained striping either by designing our applications and disk layout around them, or by using larger stripe sizes if we must. Since we know where the advantages and disadvantages of fine-grained striping are, we can draw some specific conclusions about Oracle Server.

3.3.2 High Concurrency Operations

If you are executing a high concurrency application upon a disk array (i.e., a lot of processes executing simultaneously will compete for I/O to the array), then you should ensure that your stripe size is at least large enough to allow most of your I/O calls to be serviced by exactly one disk. Otherwise, the number of physical I/O operations can grow large enough to put enormous stress on your operating system kernel and your whole I/O subsystem.

You cannot guarantee that Oracle block boundaries will align with stripe boundaries, so if you match your stripe size to your I/O call size, you will probably incur many more physical I/O operations than I/O calls. Selecting a stripe size of at least twice your expected I/O call size will give each I/O at least a 50 percent chance of requiring service from no more than one disk. In general, using a stripe size of k times your I/O size will yield a probability of $(k-1)/k$ that no more than one disk will participate in servicing an arbitrary I/O. For a given choice of k, you must ensure that your disk array can sustain $(k+1)/k$ times as many I/O calls as your application generates.

Concur-rency	I/O Size	Best Stripe Size	Oracle Server Application Examples
low	small	$k \times$ **db_block_size**, $k = 2, 3, 4, \ldots$	DBWR
low	large	$k \times$ **db_block_size**, $k = 0.25, 0.5, 1, 2, 3, \ldots$	LGWR, ARCH, single-threaded data loads and other batch processes, decision support systems (DSS) without parallel query optimization, single-threaded maintenance operations
high	small	$k \times$ **db_block_size**, $k = 2, 3, 4, \ldots$	On-line transaction processing (OLTP) systems
high	large	$k \times$ **db_block_size** $\times$ **db_file_multi-block_read_count** $k = 2, 3, 4, \ldots$	Concurrent reporting, any parallelized Oracle Server operation, other simultaneously executed high-I/O batch processes

Figure 3. Optimal stripe size as a function of concurrency level and I/O size. Striping granularity on high concurrency disk arrays should not match I/O call granularity because stripe boundaries do not necessarily align with I/O request boundaries. The recommended stripe sizes are thus k times the I/O size so that the system will fulfill each I/O request from a single drive with probability $(k-1)/k$.

Thus, if access to a disk array is exclusively high-concurrency indexed or hash-based access to Oracle data, then a stripe size equal to at least two times the value of **db_block_size** will yield excellent performance. If access to a disk array includes many sequential scans, then the optimal stripe size will be at least two times the value of **db_file_multiblock_-read_count** for those processes times the value of **db_block_size**.[7]

Fine-grained striping can be a bad choice for the tablespaces to which Oracle Server writes sort segments if a lot of applications processes will be sorting simultaneously. Fine-grained striping can also yield surprisingly bad results for applications that use the Oracle's parallel query optimization (PQO) feature, because PQO's use of multiple query slaves manufactures its own high concurrency level even for a single user session.

3.3.3 Low Concurrency Operations

If only a small number of processes compete for I/O to a striped disk array, then you have more freedom to use stripe sizes that are smaller than the size of your logical I/O calls. A low concurrency level allows you to focus on increasing single-process throughput to a disk array.

An excellent example of a low concurrency, high volume sequential writer is the Oracle Server redo log writer (LGWR). LGWR is a single-threaded processes that does high-volume sequential writes during OLTP and data-loading activity.[8] Placing on-line redo log files on a dedicated array of fine-grain striped disks yields excellent performance advantages because the whole array can be devoted to yielding the highest possible parallelized transfer rate to a single process whose speed is critical to the application.

Several more opportunities in this category generally exist in well-designed Oracle Server applications. Good Oracle Server applications batch their high-volume insert and update activity whenever possible into single-threaded, high-speed events. This is a good design technique that effectively minimizes the high concurrency phenomenon that makes fine-grained striping less attractive. A disk array that is

[7] In Oracle7.3, the value of **db_file_multiblock_read_-count** is configurable at the session level, which will yield more flexible control over the performance of applications full-table scan operations.

[8] To be more specific, LGWR only writes redo for data loads that are performed through the Oracle's standard SQL engine. LGWR does not write redo for direct-path data loads or transactions performed using the **unrecoverable** option.

fine-grain striped to maximize high-concurrency query performance will also maximize batch data loading or reporting performance if the process can be designed or scheduled so that it doesn't compete for I/O with high concurrency data access.

One final example of a large-I/O operation that can usually be single-threaded is file maintenance, like a data file restore process. A critical component of system availability is how quickly you are able to perform maintenance operations upon your database when it goes down. Restore processes can generally be designed as single-threaded as you wish, to relieve I/O contention during restore. Happily then, availability and serviceability constraints do not generally limit your ability to use fine-grained striping to maximize normal operational system performance.

3.3.4 Challenges

The hardest problem with selecting the right stripe size for a given disk array is that sometimes the usage characteristics of the array will not fit entirely into one category or another. For example, a file that is commonly accessed via high-concurrency indexed queries (motivating use of fine-grained striping for the file) may also be frequently accessed by full-table scans (motivating course-grained striping). When you run up against a really stubborn trade-off in this area of your analysis, your best answer will almost always be to use one of the following techniques.

- *Job scheduling*—For example, don't run large data loads at times when the load processes will maximize contention for I/O services with mission critical reads and writes.
- *Data placement*—Partition your database data into files that have similar I/O characteristics, and allow two files to reside on the same disk array only if those files have similar I/O characteristics. For example, it would be a horrible idea to use very fine-grained striping for on-line redo log files if you were to place other files on the array whose processes would compete with LGWR for I/O.
- *Application design*—Design single-threaded (i.e., batched), high speed, direct path data loading into your application whenever possible instead of using resource intensive transactions executed at high-concurrency. Bad transaction design not only makes it difficult to construct a good disk configuration, it also heats up Oracle Server latches and locks, transaction tables, causing a whole cascade of hard problems.
- *SQL optimization*—Sort to disk as infrequently as possible, and use full-table scan access paths as infrequently as possible. Eliminating one unnecessary sort or full-table scan from a frequently executed SQL statement can sometimes make a remarkable difference in the entire application.

Fortunately, most trade-offs can be managed quite nicely with the right mixture of (1) investment into good hardware and software, and (2) willingness to optimize the application design and use to help that hardware and software do their best work.

3.3.5 Summary

There is no single "best stripe size" for all applications, or even for all operations within a single well-defined application. You should expect to use different stripe sizes for different disk arrays in a good VLDB I/O subsystem configuration. In general, you should use the smallest stripe size that you can to eliminate disk hot spots, but you must not make your stripe sizes so small that you cause I/O thrashing on at high concurrency levels. You should use the following guidelines for choosing the stripe size for a disk array:

- *High concurrency*—If there will be high-concurrency I/O to the array, then use a stripe size that at least two times as large as the smallest of your I/O calls. For high-concurrency Oracle Server data files, this means that you should never use a stripe size smaller than $2 \times$ **db_block_size**. If these files incur frequent sequential reads, then you should ensure that your stripe size is at least $2 \times$ **db_file_multiblock_read_count** $\times$ **db_block_size**.
- *Low concurrency*—If your concurrency level is low for a particular disk array, then you may wish to use a stripe size that is smaller than the size of your I/Os to maximize throughput to the array for a small number of concurrent processes. Very fine-grained striping is good, for example, for on-line redo log files.

A simple empirical method for selecting the best stripe size for a given operation is:

1. Determine the types of I/O operations that will take place to the disk array under consideration. The most common mistake in selecting a stripe size is the failure to consider one or more key

operation that will take place upon the disk array.

2. Create several arrays with different stripe sizes that will allow you to test the different stripe sizes on your particular hardware with your particular application.
3. Perform identical operations on each array. Write-intensive tests that are easy to set up include tablespace creations, direct path data loads, and index creations. Read-intensive tests that are easy to set up include full-table scans and indexed scans.
4. Measure the performance of the tests, and choose the configuration that yields the best results. Key statistics include elapsed time, number of physical read and write calls, and for RAID 5 the number of checksum reads.

Reasonable stripe sizes for Oracle Server configurations generally range from 16–32KB for fine-grained disk arrays, to two to four times your system's maximum I/O size for coarse-grained disk arrays. The maximum Oracle Server I/O size today is 64KB on most ports and 128KB on a few ports,[9] so good coarse-grained stripe sizes for Oracle configurations generally range from 128KB to 512KB.

3.4 Array Size

Data striping yields excellent long-term uniformity of I/O across disks, which motivates us to use large disk arrays. However, the larger a disk array, the higher the outage frequency for that array becomes. Let's consider these two issues one at a time to build a method for choosing optimal disk array sizes.

3.4.1 Throughput

Disk striping, used correctly, is a superb tool for increasing disk throughput. Striping provides transparent parallelization of I/O calls across lots of relatively inexpensive disk drives. This parallelization serves large numbers of small I/O requests simultaneously, and it yields transfer rates for large I/Os that are many times higher than the rate of your fastest disk drive [Chen et al. (1993), 151].

You can calculate the minimum RAID array size required to sustain a given throughput level with the following simple technique.

1. Specify the maximum sustainable throughput capacity of each of your disk drives. This capacity figure should be the maximum *sustainable* I/O rate per disk drive.[10] Call this capacity c, expressed in I/Os per second.
2. Estimate the total number t of I/O calls that you concurrent transaction load will generate.
3. Compute the total number r of striped physical I/O operations per second that you will require of your disk array, using

   $$r = \frac{k+1}{k} t,$$

   where k is your stripe size divided by your I/O size (described above in our stripe size discussion). For example, if your application will generate $t = 300$ I/O calls per second against the array, and your array's stripe size is two times your I/O size, then your disk array must sustain 450 physical I/O operations per second.
4. Compute the minimum number of disks $g = r/c$ that must be in a disk array that is capable of sustaining this required I/O load:

   $$g = r/c = \frac{k+1}{kc} t.$$

5. Note that the transactional throughput rate requirement t (computed in step 2) may depend on the size of your disk array g (computed in step 4). For example, halving the size of your disk array may nearly halve the number of transactions that will be executed upon that array. Your disk array vendor can tell you the maximum recommended size of your brand of striped disk array.

This is a straightforward technique for computing array sizes for RAID 0 and RAID 0+1 arrays. If you use this technique to compute RAID 5 array sizes, you must increase your I/Os-per-transaction estimate for RAID 5 to reflect the high overhead cost of each write call. The influence of a disk cache, which reduces the cost of this write call overhead, makes the computation a little less straightforward.

[9] The maximum I/O size that UNIX will support is between 64KB and 512KB on most ports.

[10] This *sustainable I/O rate* figure is computed by your disk drive vendor as the highest rate that the disk drive can sustain without driving disk utilization above 60–70%. Limiting device utilization this way keeps I/O wait times within predictable boundaries.

3.4.2 Availability

Computing the right disk array size for a RAID 3 or RAID 5 configuration is a little more complicated than just computing the number of drives needed to sustain your required throughput. The additional complication is caused by the degraded performance of RAID 3 and RAID 5 configurations during outage.

Recall that RAID 3 and RAID 5 provide protection against data loss at a presumably significant acquisition cost advantage relative to RAID 1. However, what the implementers of RAID 3 or RAID 5 sacrifice in return for this cost savings is significantly increased vulnerability to performance degradation during outage. We have also seen that one way to reduce the cost of a RAID 3 or RAID 5 configuration is to increase the size of the disk array, but that this savings comes at the cost of higher outage frequency. Consequently, assessing your aversion to performance penalties induced by partial disk subsystem outage is an important step in calculating your best RAID 3 or RAID 5 array size.

RAID 1 and RAID 0+1 inflict no performance penalty during outage of a disk protected by an operating mirror disk. The computation procedure for optimal RAID 1 and RAID 0+1 array sizes is thus unencumbered by similar considerations.

3.4.3 Summary

There is no single “best disk array size” for all applications. Larger array sizes yield better throughput performance at the cost of higher outage frequency for the array. Just as a well-designed system may contain different RAID configurations on different arrays, a well-designed system may contain different array sizes for different disk arrays, depending upon the specific characteristics of the data being stored in each array.

For RAID 1 and RAID 0+1 configurations, larger array sizes improve performance without reduction in fault resilience. For RAID 3 and RAID 5 configurations, larger array sizes improve performance but penalize fault resilience.

3.5 Raw Devices

Raw devices are an important tool that the VLDB architect uses to reduce CPU load for write-intensive applications. A *raw device* is an unformatted UNIX disk slice that Oracle Server can open as a data file or an on-line redo log file without using the standard UNIX buffered I/O services. Oracle Server’s ability to bypass the overhead of UNIX buffering reduces the operating system code path for write operations. Raw devices are thus advised for well-designed VLDBs with high transactional throughput requirements.

Raw devices are generally required today if you intend to use Oracle Parallel Server for UNIX. Most UNIX implementations do not yet allow two cluster nodes to access mounted file systems simultaneously.

Raw devices incur operational costs beyond those of administering UNIX file systems (ufs) [Millsap (1995a), 15–17]. But for write-intensive VLDB, these costs pale in comparison to the combined costs of unnecessary CPU overhead and the already significant costs of administering a system with hundreds or thousands of disk drives.

- *Random read performance*—Marginal to negligible improvement compared to ufs.
- *Random write performance*—Excellent improvement compared to ufs because of code path reduction. Raw devices also permit implementation of asynchronous I/O if the platform offers it.
- *Sequential read performance*—Marginal performance degradation compared to ufs. Using raw devices can dramatically degrade the performance of poorly optimized SQL applications compared to ufs implementations because UNIX buffer caching outperforms Oracle Server caching for full-table scans.
- *Sequential write performance*—Excellent improvement compared to ufs because of code path reduction and async I/O capability.
- *Outage frequency*—Increased risk due to the more sophisticated administration talent required to administer raw devices.
- *Outage duration*—Increased risk due to the more sophisticated administration talent required.
- *Performance degradation during outage*—Difference from ufs is isolated to the normal operational performance characteristics described above.
- *Acquisition cost*—Same as ufs.
- *Operational cost*—Worse than ufs. There are large training and salary costs associated staff competent to configure and maintain raw disk. Raw device configurations also require purchase or development of custom software tools to help

manage the I/O subsystem. Differential costs of administering raw versus ufs are negligible as a proportion of total I/O management spending in a VLDB environment with thousands of disks. No incremental cost of using raw devices for on-line redo log files because these files are backed up the same as ufs by the Oracle ARCH process.

3.5.1 Techniques

The VLDB architect should employ the following techniques in implementing raw devices:

- *Standard redo log file sizes*—Configure raw slices that will be used for on-line redo log files to be all the same size.
- *Standard data files sizes*—Use one slice size for all raw slices to permit you to move the contents of a raw slice from one disk to another if you need. Standard raw slice sizes are also an important component of a good Oracle storage parameter management plan. If your database contains one or more small tablespaces for which large raw slices would be wasteful, you may wish to select one or two more standard raw slice sizes for those tablespaces. If you use more than one raw slice size, then make the sizes integral multiples and divisors of each other.
- *Keep some slices unallocated*—Leave some raw slices unallocated so that you will have spare disk capacity to handle database growth. Oracle Server gives you the capability to **alter tablespace … add datafile** and **alter database add logfile**, but you can't use either if you don't have spare slices in the right sizes that are available to allocate.

3.6 Oracle Configurations

There is no such thing as a single optimal Oracle configuration for all VLDB applications. Your optimal mixture of *fast*, *reliable*, and *inexpensive* depends upon your specific goals. However, we can make some general conclusions about how you would optimize your Oracle Server VLDB configuration if you were unencumbered by economic constraints:

Control files
raw
n-way multiplexed RAID 0+1
fine-grain striped
stored on dedicated disks

On-line redo log files
all files the same size
raw
n-way multiplexed RAID 0+1
fine-grain striped
stored on dedicated disks

Archived redo log files
ufs (because you have to)
n-way multiplexed RAID 0+1
fine-grain striped
stored on dedicated disks
separated from on-line redo log files

Data files
all files one of 1–3 standard sizes
raw if
    file is write-intensive, or
    instance is Oracle Parallel Server
ufs if
    segment is full-table scanned
n-way multiplexed RAID 0+1
stripe size at least 2× I/O size if
    I/O concurrency to the array is high
stripe size less than 2× I/O size only if
    I/O concurrency to the array is low
RAID 5 for an array only if
    I/O is not write-intensive

Other files
ufs
n-way multiplexed RAID 1 if possible
striped if I/O character indicates
named, distributed per *OFA Standard*

If you are cost-constrained, then you must evaluate your less expensive options for the configuration that yields the most value to you per unit of investment.

3.6.1 Example Configurations

You can mix and match technologies in a large number of ways. Three sample Oracle Server configurations are defined in Figure 4. Each represents a different level of investment required to meet the specific goals of a specific implementation.

- *Configuration A*—Multiplexed ($n = 2, 3$) RAID 0+1 for all files, raw devices everywhere possible. Mission critical, OPS-ready, very high transaction rate, very high query rate, operationally complex, high cost.
- *Configuration B*—Raw RAID 0+1 for write-intensive files, ufs RAID 5 and ufs for infrequently updated files. Moderately high availability, non-OPS, moderate transaction rate, very high query rate, moderate operational complexity, moderate cost.

File Type	Configuration		
	A	B	C
On-line redo log files	raw 0+1	raw 0+1	raw 0+1
system, temp, rbs	raw 0+1	raw 0+1	ufs 0
Other data files	raw 0+1	ufs 5	ufs 0
Archived redo log files	ufs 0+1	ufs 0+1	ufs 0+1
Control files	ufs 0+1	ufs 0+1	ufs 0+1
Other files	ufs 0+1	ufs 5	ufs 0

Figure 4. Sample disk configurations for Oracle Server applications. This table describes three sample configurations using mixtures of raw and ufs disk and different RAID levels to achieve different business goals. The goals and benefits of configurations A, B, and C are described in the text.

- *Configuration C*—Raw RAID 0+1 for write-intensive files, ufs RAID 0 for all other Oracle files. Mission non-critical, non-OPS, moderate transaction rate, very high query rate, reduced operational complexity, reduced cost.

4. Redo Log Files

Oracle Server uses a dedicated process called the *redo log writer* (LGWR) to write batches of small transaction entries to disk. The actions of LGWR allow Oracle Server to write committed blocks to disk asynchronously with respect to user **commit** requests and still maintain data integrity even if the server fails at an awkward moment. Dedication of a background process to serialize and batch writes of change vectors to disk is an enormous performance advantage of the Oracle Server Architecture.

LGWR performance is critical to the performance of OLTP systems and data loading procedures that do not use Oracle Server unrecoverable transaction features. Environments that do not write intensively through the standard SQL engine to the database do not stress LGWR. Especially in high-volume OLTP systems, badly configured redo logging will degrade either performance or outage duration or both; hence, configuring redo logs is the focus of this section.

4.1 Performance/Availability Trade-Offs

The battle between performance and reliability meets full strength at the redo log file. To optimize performance, you will configure your LGWR and DBWR to write as *in*frequently as possible. However, to minimize the roll-forward time associated with instance recovery, you will configure your LGWR and DBWR to write as *frequently* as possible without treading on your performance requirements. The challenge is thus to know your technology well enough to find a value of **log_checkpoint_interval** that is simultaneously large enough and small enough to meet both requirements.

Understanding your technology relative to redo requires understanding of two key events: Oracle Server checkpoints, and Oracle Server instance recovery.

4.1.1 Oracle Server Checkpoints

An Oracle Server *checkpoint* is an event begun when LGWR signals DBWR to write all the modified blocks in the database buffer cache, including committed and uncommitted data, to the data files [Concepts (1992), 23.9–12]. Checkpoints induce a brief load spike on the system, the effects of which the database administrator tries to minimize through tuning [Admin (1992), 24.6–7]. Normal operational checkpoints occur at the following times:

- *Log switch*—When LGWR fills a redo log file and attempts to switch to the next one in the circular queue, LGWR issues a checkpoint.
- *Predefined interval*—LGWR will issue a checkpoint whenever **log_checkpoint_interval** operating system blocks have been written to disk since the most recent checkpoint. LGWR will also issue a checkpoint whenever **log_checkpoint_timeout** seconds have passed since previous checkpoint began.

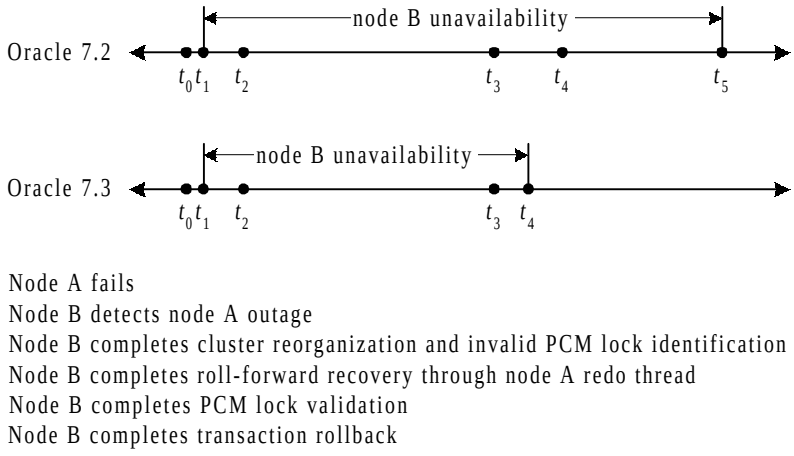


Figure 5. Failover timelines for Oracle Parallel Server releases 7.2 and 7.3. OPS 7.3 defers rollback application to minimize the duration of cluster unavailability during node failover, yielding better node recovery performance than OPS 7.2 and prior versions. In OPS 7.3, the time at which instance-wide rollback is complete does not matter to an individual application since all 7.3 rollback is executed on an as-needed, transaction-by-transaction basis.

Redo log file size and the values of **log_checkpoint_interval** and **log_checkpoint_timeout** are the most important influencers of performance degradation from normal operational checkpoints. Many database administrators deactivate the time-out by setting it to zero, and they control checkpoint frequency with the **log_checkpoint_interval** parameter. Some DBAs simply disable the timeout and use such a large checkpoint interval that no checkpoints occur other than those motivated by log switches.

4.1.2 Oracle Server Instance Recovery

The amount of time required for Oracle Server instance recovery is important to the VLDB architect because it defines the application outage duration for the following events:

- *Unclustered server node failure*—Failure of a CPU, bus, or memory board in a non-clustered Oracle Server configuration causes application outage.
- *Oracle Parallel Server node failure*—Failure of one or more Oracle Parallel Server nodes will cause a brief cluster-wide outage that ends when a surviving node completes instance recovery on the failed node's redo thread.

The Oracle7.3 implementation of on-demand deferred roll-back makes the cluster unavailability period dependent predominantly on the time needed to reconstruct the database buffer cache from the failed node's redo thread. The timeline for Oracle7.3 failover compared to Oracle7.2 failover is shown in Figure 5.

Oracle Server instance recovery is a tunable process whose time consumption depends predominantly on two factors [Maulik and Patkar (1995)]:

- The amount of redo that has been generated since the most recent checkpoint (less is better); and
- The cache hit ratio on the database buffer cache during recovery (higher is better).

The key influencers over the first factor are the system's redo generation rate and the checkpoint frequency. Reducing total redo generation is an application design task of minimizing the weights of your transactions, minimizing the number of transactions your system must execute to meet your business goals, and exploiting Oracle Server's unrecoverable transaction features wherever possible. Reducing the checkpoint frequency is a simple matter of adjusting the checkpoint interval or perhaps increasing redo log file size.

You can improve the second factor by increasing the size of the database buffer cache and by reducing the likelihood of cache misses during the recovery process. Increasing the setting of **db_block_buffers** for the recovery process (at the expense of reducing **shared_pool_size** if you must) generally reduces database buffer page faulting. During instance recovery that does not require media recovery, you have no other control over the recovery process cache hit ratio because your roll-forward must execute given the state of the system at the most recent checkpoint before the outage.[11]

Settings of **log_checkpoint_interval** that yield generally good trade-off performance in OLTP environments range from a few kilobytes to a few hundred megabytes (specified in operating system blocks). Roll-forward recovery rates seem to vary almost as dramatically as roll-forward recovery rate requirements.[12] For environments with very stringent performance and reliability requirements, small changes to **log_checkpoint_interval** can make big differences in instance recovery time. Consequently, finding the optimal setting of **log_checkpoint_interval** frequently requires specific testing for the environment being used.

4.2 Redo Log File Placement

A good VLDB configuration must isolate the high-I/O redo log files as much as possible from all other high-I/O files. You maximize your chances of preventing log file I/O bottlenecks if you isolate your redo log files on disks and controllers dedicated to serving only redo log files. This isolation technique meshes well with our desire to have redo log files on a fine-grain striped disk array that is accessed by a single-threaded (very low concurrency level) process.

In addition to separating log files from everything else, you in fact must separate log files from each other. While LGWR is writing to an on-line log file as fast as it can, ARCH is reading an on-line log file as fast as it can (at a given moment, it could be reading any one of the on-line log files except for the one currently held open by LGWR), and ARCH is simultaneously writing as fast as it can to one of the archived redo log files. Fine-grained redo log file striping can minimize the contention between LGWR and ARCH.

Oracle Server allows you to archive redo log files directly to tape if you want to. There are several reasons that VLDB database administrators almost always archive first to disk and then copy the archive files to tape with *n*-way multiplexed writes.

- *Better reliability*—Tapes are one or more orders of magnitude less reliable than disks. Entrusting one's only surviving copy of a potentially mission-critical archived redo log file to a tape is not very smart.
- *Better throughput*—If you have ever run **truss** on the archiver process, you know that ARCH is not simply a streamed copy process, but that the operation performs best on a device capable of random I/O. Archiving to disk is much faster than archiving to tape, which reduces the likelihood that ARCH will become an unnecessary bottleneck.
- *Better error control*—You can much more reliably manage disks that fill than tapes that fill. Space management can be software automated for disks, but the process requires manual intervention for tapes.
- *Faster restore times*—Many database administrators keep on disk as many archived redo log files as would be required to recover the tablespace with the oldest current hot backup. This technique reduces restore process duration by the amount of time required to find the right tape, mount it, and read the archived redo log files from the tape to disk.

4.3 Redo Log File Size

A system's optimal redo log file size depends on redo generation rate and its instance recovery outage duration requirement. A sound method for calculating the right log checkpoint interval and log file size for your system is:

1. Specify your crash recovery time requirement. Call this time *t*, expressed in seconds.
2. Compute the rate at which your system applies redo in the instance recovery process. Call this rate *r*, expressed in bytes per second.

[11] Maulik and Patkar's results are important for Oracle Server *media* recovery from multiple data file corruptions, such as results from user input error. They recommend executing multiple *restore-file/recover-instance* cycles if recovery from several data files is required, and if the database buffer cache hit ratio during each recovery pass can be improved by doing so [Maulik and Patkar (1995), 7.25].

[12] Oracle database administrators have reported wide-ranging roll-forward rates. Recently, I've heard reports as low as 50KB/sec and as high as 1,750KB/sec.

3. Set the value of **log_checkpoint_interval** to rt/b, where b is your operating system block size in bytes.
4. Create your redo log files with size $f = krt$, for some $k = 1, 2, 3, \ldots$ (i.e., f is an integral multiple of rt).
5. If checkpoints begin to accumulate without completing, then you must increase the value of **log_checkpoint_interval**. You can determine whether you are checkpointing too frequently by comparing *background checkpoints started* with *background checkpoints completed* in **v$sysstat**. If the values differ by more than one, then your checkpoints are not completing before newer checkpoints are started.
6. Choose a **log_buffer** size of $b = 3ns$, where n is the number of disks in the disk array holding the on-line redo log files, and s is the disk array's stripe size in bytes. The largest write that LGWR will generally execute is roughly $b/3 = ns$.[13] Using $b = 3ns$ then means that LGWR will seldom write more than one stripe per disk in the disk array holding on-line log files.

This method reduces the task of choosing your optimal redo log file size to the task of choosing the appropriate integral multiple k of your checkpoint interval size. Factors that should influence you to use larger values of k include:

- reduced redo log switch frequency
- simplified redo log file placement decisions
- reduced media recovery complexity because there are fewer files to handle
- reduced frequency of checkpoint busy waits and archiver busy waits

Factors that should influence you to use smaller values of k include:

- reduced outage cost associated with losing all copies of your active on-line redo log file[14]
- improved flexibility in preventing *file system full* errors on the archive destination device
- reduced disaster-time data loss in standby database configurations

A general rule of thumb in sizing redo log files is that you should attempt to log switch no more frequently than about twice an hour.

4.4 Redo Log File Quantity

The number of on-line redo log files you create helps determine whether checkpointing and archiving will cause transaction processing bottlenecks. Selecting the right number of redo log files helps to reduce:

- *Checkpoint busy waits*—A checkpoint busy wait occurs when LGWR attempts to log switch into a log file before the checkpoint associated with the switch out of that file has had a chance to complete.
- *Archiver busy waits*—An archiver busy wait occurs when LGWR attempts to log switch into a log file whose contents have not been successfully copied to an archived redo log file by ARCH.

Checkpoint busy waits were frequent occurrences in the days when most Oracle Server implementations used the default installer configuration of only two on-line redo log files. You can use the following process to detect and fix checkpoint busy waits:

1. Check **v$session_wait** for *log file switch (checkpoint incomplete)* events. If you have **log_checkpoints_to_alert** set to **true**, then you can also search for the string "cannot allocate" in the **alert.log** file.
2. You can reduce checkpoint busy waits by
   - increasing the number of on-line redo log files to reduce the probability that LGWR will wrap through the circular queue of redo log files before a checkpoint has time to complete; or

[13] In OLTP operations, frequent transaction commits will trigger frequent LGWR writes, and LGWR will make very small write calls. In batch operations, infrequent commits allow large amounts of unwritten redo to build up in the log buffer. If this redo accumulates quickly enough, even the event that triggers LGWR to write when its buffer is 1/3 full can motivate write calls of up to b bytes, where b is the setting of **log_buffer**.

[14] You should try to protect yourself from this catastrophic event by using n-way multiplexed hardware for storing on-line redo log files. If you can't afford a multiplexed hardware solution, Oracle7 allows you to mirror in software by using multiple redo log file members in a log file group.

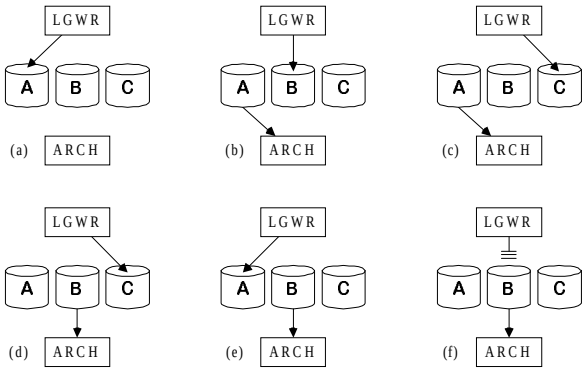


Figure 6. An archiver busy wait. This figure shows how a fast LGWR process can overtake a slower ARCH process to cause a performance bottleneck. Panel (a) represents the instance immediately after startup, with LGWR writing to file *A* and ARCH not yet active. In (b), LGWR has switched to file *B*, allowing ARCH to open *A* for reading. In (c), LGWR has switched to file *C*, but ARCH has not yet completed its reading of *A*. In (d), ARCH has switched to reading *B*, but in (e), LGWR has switched into *A*. In (d), lgwr attempts to switch to file *B*, but it cannot because the file is currently being archived. At this point, ARCH will prevent the completion of any **commit** requests until it switches into file *C*. If the redo generating process that is keeping LGWR so busy lasts much longer, then ARCH will persist in blocking transaction processing on the system.

- adding DBWR processes to increase the speed of checkpoints (only if you're using synchronous writes); or
- increasing the value of **db_block_checkpoint_batch** to increase the speed of checkpoints; or
- reducing the value of **db_block_buffers** to reduce the size of a checkpoint; or
- reducing the size and number of rollback segments in the database (described in a later section).

Archiver busy waits typically occur during high-volume transaction bursts in batch processing, when the writing of redo by LGWR outpaces ARCH's ability to keep up. This causes log switch waits to be gated on the ARCH process—ARCH becomes the bottleneck for all **commit** processing in the system.

Techniques for alleviating this problem include:

- *Fine-grained striping*—Place your archived redo log files on a disk array that uses fine-grained striping to maximize ARCH throughput to the files.
- *More on-line redo log files*—Create enough on-line redo log files that LGWR can't wrap around to catch ARCH from behind before the burst is finished. Think of a foot-race on a standard oval track. One way to prevent slow runners from being passed (i.e., "lapped") by faster runners is to make the track so large that the race ends before the first-place finisher has a chance to pass the last-place finisher.
- *Multiple archivers*—Use multiple ARCH processes with Oracle Server's **alter system archive log all to …** syntax. Oracle Services consultants have prepared an assortment of UNIX tools to automate this process.

5. Tablespace Partitioning

When you convert your logical data model into a physical data model, you are required to make long-term decisions about how you will partition your database segments into tablespaces. Constraints that determine how you should partition your segments into tablespaces include the following:

- *I/O performance*—Each tablespace should contain segments whose I/O concurrency and I/O size characteristics are similar, to facilitate disk array size and stripe size selection. Grouping read-only segments into read-only tablespaces reduces backup and recovery data transfer volumes and reduces PCM lock maintenance.

Separating frequently written segments from infrequently written segments yields flexibility in hot backup procedure construction.

- *Outage resilience*—Small tablespaces allow off-line tablespace maintenance with minimal application outage. The **system** tablespace cannot be taken off-line, and it cannot be dropped and recreated, so keeping as few segments there as possible minimizes the need for database down-time. Isolating rollback segments into as few tablespaces as possible reduces outage frequency, because any tablespace containing an on-line rollback segment cannot be taken off-line. Storing referentially related segments in small groups of tablespaces maximizes your ability to exploit the tablespace point-in-time recovery feature scheduled for Oracle8.
- *Space management*—Isolating segments with short lifespans minimizes the impact of tablespace free space fragmentation that can block Oracle Server extent allocation. VLDB administrators profit from using tablespace default storage parameters instead of maintaining segment sizing parameters at the segment level.
- *Quota management*—Oracle Server space quotas are administered to users by tablespace. Hence, you should assign groups of segments in one schema to a small group of tablespaces.

5.1 Assigning Segments to Tablespaces

The following method will help you make good decisions about how to partition your Oracle segments into tablespaces.

1. Put only dictionary segments (segments owned by **sys**) in the **system** tablespace. Drop the **aud$** table and create it in a tablespace other than **system** so that you can administer to this table's growth and shrinkage without fragmenting the **system** tablespace. Some experts recommend editing **sql.bsq** (only lines below the // token in the file) to modify the location and storage parameters for the dictionary tables containing stored procedures and triggers. Secure the full cooperation of your Oracle Worldwide Support representative before you attempt to edit **sql.bsq**.
2. Create two or more tablespaces devoted exclusively to temporary segments. Create a program that will allow you to switch your users' temporary tablespace settings quickly to the names of on-line tablespaces other than **system** in the event of temporary tablespace media outage [To (1995), 6.12].
3. Create one or more tablespaces devoted exclusively to rollback segments. Do not place a rollback segment in any tablespace other than one designed exclusively for rollback segments.
4. Isolate transient application segments with short lifespans in as few tablespaces as possible. Do not place a transient application table or index in any tablespace other than one designed exclusively for transient segments.
5. Exploit every opportunity to isolate read-only segments into tablespaces that contain read-only segments exclusively. Then operate these tablespaces in read-only mode.
6. Categorize your remaining segments by size. Store only similar-sized segments in a given tablespace.
7. Limit the maximum size of a tablespace to about 10GB. Small tablespaces incurring file outage can be taken off-line with potentially limited impact to database availability. Large tablespaces are much more likely to cause database application outage for a file outage event. Using small tablespaces will enable parallelization when tablespace point-in-time recovery becomes available in a future release of Oracle8.
8. If you are using fine-grained striping for data files, then it is not necessary to separate indexes from data to distribute I/O load across disks. However, if your operational procedures include periodic index rebuilds, you may want to consider isolating indexes into their own tablespaces to minimize the possible impact of tablespace free space fragmentation caused by the **drop index**.

6. Storage Parameters

Storage parameters have long been a hot topic because many Oracle DBAs believed that it was important to try to make each of their database segments use no more than one extent. More and more people have learned that multiple extents bear negligible performance impact upon all Oracle DML operations. The time spent "compressing" tables and

indexes into one extent was generally time that could have been spent elsewhere for greater gain.[15]

6.1 Maxextents

The real reason that DBAs must pay attention to storage parameters is that the maximum allowed value of **maxextents** is limited in pre-7.3 releases by the value of **db_block_size**. The threat that an application could fail with a *max # extents reached* error focused database administrators' attention on fitting their segments into a limited number of extents. With the **maxextents unlimited** capability of Oracle Server Release 7.3, this **maxextents** threat goes away.

Ironically, with VLDB, storage parameters are important for precisely the opposite of the original historical reason. With VLDB, there are several cases in which you *want* your segments to be stored in more than one extent.

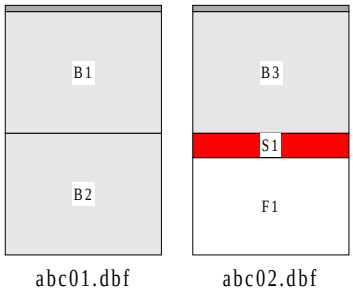
- Using multiple extents allows reclamation of tablespace free space with the **truncate … drop storage** command.
- You *want* rollback segments to have multiple extents to minimize system load generated by dynamic extent allocation and deallocation [Millsap (1995b)].
- You want the ability to purchase disk hardware as your system grows, and Oracle's ability to grow segments dynamically as required suits this goal.
- Proper use of multiple extents helps reduce false pinging in Oracle Parallel Server implementations.
- You *want* tablespace pre-fragmentation in any tablespace containing temporary segments or rollback segments, to minimize extent allocation overhead.

6.2 Storage Parameter Attributes

Good Oracle Server VLDB storage parameters have the following attributes:

- *Database block size multiples*—All extent sizes are integral multiples of the database block size. Oracle will always round extent sizes to whole blocks, so you might as well ask for the size you're going to get. For example, **10K** is a silly value for **initial** in a database with 8KB Oracle blocks. Initial extents in an 8KB-block database will be allocated at least 16KB of space, regardless of what you ask for.
- *File size divisors*—All extent sizes should be integral divisors of your usable data file size. This enables you to fit extents nicely into data files with no space waste.
- *Small number of extent sizes*—The number of distinct extent sizes in a given tablespace should be small; for example, only one. If there is more than one extent size in a tablespace, then all those extent sizes are integral multiples or divisors of each other.

The picture below shows two data files with two extents allocated within each. The tiny region at the beginning of each data file represents the data file overhead associated with each data file.[16]



In this example, the big extents *B1*, *B2*, and *B3* are sized so that each data file can contain exactly two extents with no space waste. The data file called **abc01.dbf** contains exactly two of these big extents, which fill the file nicely. In **abc02.dbf**, Oracle Server allocated a small extent, denoted here as *S1*, in the same datafile with the big extent *B3*. The remaining free space, labeled *F1* here, is quite large—almost half the data file—but not large enough to contain a standard large extent the size of *B1*, *B2*, and *B3*.

The *S1* extent is a space waste irritant because its size is different from the size of *B3*, which left the huge wasted chunk of free space (*F1*) in the tablespace. The fact that the size of *S1* is not an integral divisor of the size of *B3* means that *F1* is an odd size that makes it even more difficult for another extent to use all of the free space in **abc02.dbf**.

[15] As people learn this, they discover that if they can get better DML performance from "compressing" the segment, they could have gotten the same improvement by better managing the multi-extent segment [Millsap (1995b)].

To understand the true impact of this type of problem on VLDBs, you need to understand that the number of data files in an Oracle7 VLDB is bound to be several hundred, and in Oracle8, this number will grow to several thousand. A problem that requires one hour of database administrator time in a 5GB database with 20 data files will require 30 hours or more to repair in a 100GB database with 600 data files. The combination of bigger problems and tougher availability requirements in VLDB environments will completely overwhelm a database administrator unless you can implement standards like the ones we're discussing here.

In tablespaces containing smaller segments, you may want to use more than one extent size (to preserve space, to reduce the number of data files in your database, to reduce the number of changes to your prepackaged applications, etc.). If you must use more than one extent size in a tablespace, use extent sizes that are multiples and divisors of each other to maximize the reusability of your space.

If you must administer storage parameters for individual database segments, you should use the Oracle Designer/2000 storage management capabilities, or you can buy or build an Oracle Forms or Oracle Power Objects based application to administer storage parameters. A good tool will keep you out of SQL*Plus. It will allow you to assign a name to each distinct combination of **initial**, **next**, and **pctincrease** values and manipulate storage parameters by name instead of by value.

6.3 Selecting Storage Parameters

The following procedure will help you to choose storage parameters for your VLDB that suit the constraints described above.

1. Compute the size of the usable space available in your data files (choosing standard sizes for your data files, as recommended earlier, simplifies this task). The most direct way to do this is to query the **dba_free_space** dictionary view immediately after creation of your data files.

   ```
   select
     blocks, bytes
   from
     dba_free_space
   where
     file_name='filename'
   ```

   You can forecast the size of your usable data file space if you know your data file size by using a table like the one shown below.

Oracle Blocks	Bytes	Description
1	8,192	Oracle database block size
262,144	2,147,483,648	data file size reported by the O/S
2		overhead per data file (ufs=1, raw=2)
262,142	2,147,467,264	usable data file size

   If you use this model, validate your forecasts with the **dba_free_space** query shown above.

2. For each tablespace, set the default storage parameters to the following values:

   - Set **initial** to a number that is both an integral multiple of the database block size and an integral divisor of your usable file size.
   - Set **next** equal to the value of **initial**.
   - Set **pctincrease** to **0**.

   The tables below show two different sets of storage parameters that fit these constraints. Each storage parameter set contains a two-block extent size to use for very small reference tables, and the remaining storage parameters are computed by calculating the extent sizes required to split the data file into *k* equal-sized chunks. The differences in these two tables are in the numbers chosen for *k*. One uses powers of four, and the other uses powers of ten.

n	$k = 4^n$	Extent Size in Oracle Blocks	**initial**, **next** Value
*	131,071	2	16,384
8	32,768	7	57,344
7	16,384	15	122,880
6	4,096	63	516,096
5	1,024	255	2,088,960
4	256	1,023	8,380,416
3	64	4,095	33,546,240
2	16	16,383	134,209,536
1	4	65,535	536,862,720
0	1	262,142	2,147,467,264

[16] The amount of data file overhead is one Oracle block if you're using the UNIX file system; it's two Oracle blocks if you're using raw devices.

n	$k = 10^n$	Extent Size in Oracle Blocks	**initial**, **next** Value
*	131,071	2	16,384
4	10,000	26	212,992
3	1,000	262	2,146,304
2	100	2,621	21,471,232
1	10	26,214	214,745,088
0	1	262,142	2,147,467,264

3. Allow segments to inherit their default storage parameters whenever possible. Optimally, there will be only one distinct combination of **initial**, **next**, and **pctincrease** storage parameter values in each tablespace. For each segment not allowed to use its tablespace's default storage parameters, set individual storage parameters as follows:
   - Set **initial** to a number that is both an integral multiple of the database block size and an integral divisor of your usable file size.
   - Set **next** equal to the value of **initial**.
   - Set **pctincrease** to **0** for large segments, and to **0** or **100** for smaller segments.
4. For each segment, set the other storage parameters, such as **pctfree**, **pctused**, **initrans**, and **maxtrans**, individually.

If you were to use only storage parameters from the $k = 10^n$ table in a database with the file size shown earlier, then you would find only six distinct storage parameter sets in the entire database:

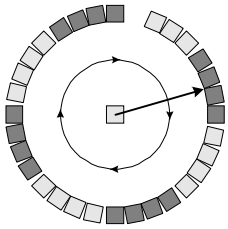
```
select distinct (
  initial_extent||','||
  next_extent||','||
  pct_increase
) "initial,next,pctincrease"
from dba_segments

initial,next,pctincrease
------------------------
16384,16384,0
212992,212992,0
2146304,2146304,0
21471232,21471232,0
214745088,214745088,0
2147467264,2147467264,0
```

7. Rollback Segments

Rollback segments are data structures that Oracle Server uses to provide the following services:

- *Read consistency*—Rollback segments contain transaction undo information. Oracle Server uses undo to construct read consistent queries on blocks that are undergoing active on-line modification during the query.
- *In-flight transaction rollback*—The same undo information is what allows Oracle Server to provide data integrity after unscheduled instance outage and user-requested roll-back of uncompleted transactions.

A rollback segment is a circular queue of Oracle database blocks into which foreground Oracle Server processes write undo entries during transaction execution. The following drawing shows an Oracle rollback segment with eight extents, each of which contains four Oracle database blocks. The block shown in the center of the drawing is the rollback segment header block. The arrow pointing to the beginning of the third block in the second extent denotes the location to which the next undo write will take place in this rollback segment. This so-called "write pointer" sweeps the rollback segment in a clockwise direction, reusing rollback segment blocks as the pointer wraps around.



7.1 Rollback Segment Size

The key constraints for sizing rollback segments properly are:

- *Small enough to cache*—Rollback segment blocks page in and out of the SGA by the same LRU rules that govern other database blocks. Large rollback segments dirty large portions of the database block buffer cache, and they flush out large numbers of other blocks (like index branch blocks) that improve application performance. The increased buffer cache page faulting incurred by large rollback segments not only reduces your system's cache hit ratio, it also dramatically increases the CPU and I/O loads generated by checkpoints.
- *Large enough for large transactions*—If your application executes large transactions that generate hundreds of kilobytes of undo between

commits, then you're stuck having at least one rollback segment on your system large enough to hold the undo generated by your largest transaction. Otherwise, the transaction will fail.

Oracle7 provides two capabilities that dramatically reduce the complexity of rollback segment administration compared to some of the hard work we once did with Oracle6: (1) rollback segment growth and shrinkage, and (2) the ability to take a rollback segment on-line or off-line without halting the instance. Even with these features, a big database administration challenge for some people remains the maintenance of rollback segments for mixed OLTP and batch processing. The key tools for minimizing negative impacts of rollback segment maintenance upon performance are:

- *Application design*—Design programs that do large amounts of DML (especially inserts) so that they exploit Oracle7.3's unrecoverable DML capabilities wherever possible. For transactions that must be executed through the standard SQL engine, design your application to perform frequent interim commits. If you can't do this, then at least direct large transactions' undo to a specifically denoted large rollback segment using the **set transaction use rollback segment** syntax.
- *Job scheduling*—If you must run batch programs that do a lot of DML without interim commits (e.g., with a pre-packaged application that you cannot customize), then schedule your batch jobs so that their undo generation will not compete with high concurrency activity in the SGA. To minimize consistent-mode read performance overhead and to prevent *snapshot too old* errors, you must also schedule long-running query jobs to run at times when concurrent DML activity is not occurring on the queried segments.

You can use the following method to size your rollback segments optimally for your system.

1. Make all of your on-line rollback segments the same size. You may wish to keep a few specially sized rollback segments off-line to use in special time windows dedicated to batch processing. If you do this, then at the beginning of the batch window, you will take your small OLTP rollback segments off-line and bring your batch rollback segments on-line. You'll run your batch and then reverse the setup process to prepare your database for the next window's OLTP activity.
2. Select the rollback segment size that is just large enough to contain the undo for some small number k of simultaneous executions of the largest transaction that you will execute in the window when this rollback segment is on-line. The criterion for selecting k is to minimize the number of blocks in the buffer cache dedicated to rollback segment storage. If your largest transaction generates less than about a database block of undo, then choose $k = 4, 5, \ldots, 10$. If your largest transaction generates hundreds of kilobytes of undo, then choose $k < 4$, and have a talk with the people who designed your application.
3. Set the **minextents** value between **8** and **20** for each rollback segment, and set **optimal** to the appropriate value to ensure that rollback segments do not shrink below eight (or twenty) extents. Note that having this large a value of **minextents** implies that your rollback segment extents will be very small. Using multiple extents in rollback segments reduces the frequency of segment growth and shrinkage events [Millsap (1995b)].

7.2 Rollback Segment Quantity

The constraints on rollback segment quantity are similar to the constraints on rollback segment size:

- *Few enough to remain cached*—Having too many rollback segments has the same ill effects on SGA page faulting and checkpoint processing as having rollback segments that are too large.
- *Many enough to avoid contention*—Having too few rollback segments causes contention on the Oracle Server *transaction table* that is stored in a rollback segment's header block. You can detect rollback segment contention by finding a non-zero number of *undo header waits* in the **v$waitstat** dynamic performance table.

Thus, you must create enough rollback segments to prevent undo header waits, but never more than your instance's maximum number of concurrently active transactions. You can estimate the number of rollback segments your system will require if you do not yet have operational measurements on the application, by using the following process.

1. Choose a confidence interval C, where $0 < C < 1$, for your prediction. For example, if you want a 90-percentile estimate, then choose $C = 0.90$.

$n = 170, \quad p = 0.05$		
x	$P(X = x)$	$P(X \le x)$
0	0.000	0.000
1	0.001	0.002
2	0.006	0.008
3	0.019	0.027
4	0.042	0.069
5	0.074	0.143
6	0.106	0.249
7	0.131	0.381
8	0.141	0.521
9	0.133	0.655
10	0.113	0.768
11	0.086	0.854
12	0.060	0.915
13	0.039	0.953
14	0.023	0.976
15	0.012	0.988
16	0.006	0.995
17	0.003	0.998
18	0.001	0.999
19	0.001	1.000
Total	1.000	

Figure 7. Estimating the required number of online rollback segments for OLTP. In this example, if we have 170 concurrent users, each of whom executes one 3-second transaction once a minute, then having 12 rollback segments will provide one rollback segment per concurrently active transaction 90% of the time. This table was created in Excel using the **binomdist** function.

2. Estimate the expected number of active users on the system at a peak load time. Call this number *n*.
3. Estimate the probability that any one transaction will be active at an arbitrarily chosen moment. Call this probability *p*.
4. Find the smallest value of *x* for which

   $$P(X \le x) > C,$$

   where

   $$P(X \le x) = \sum_{k=1}^{x} P(X = k),$$

   and $P(X = x)$ is Bernoulli's binomial distribution function, defined as

   $$P(X = x) = \frac{n!}{x!(n-x)!} p^x (1-p)^{n-x}.$$

   If you have Excel or a similar tool, you can find the value of *x* very easily, as shown in Figure 7.
5. After your system operates for a while, you can fine-tune your number of rollback segments by monitoring **v$waitstat**. If you have *undo header waits*, you can solve the problem by adding rollback segments. If you do not have undo header waits, then you may be able to drop one or more rollback segments without impacting system performance.

8. Conclusions

To implement Oracle Server successfully for very large databases, you must know your requirements, and you must know your technology. In this paper, I have identified several configuration issues that can prevent a system from ever achieving the expectations that have been placed upon it. At the same time, I have tried to provide a structured way of thinking about technology to highlight the trade-off decisions that ultimately determine what you really want your requirements to be.

There is no single perfect general configuration for VLDB, because different owners of different systems will always have different goals and priorities. The contribution I have tried to make is to identify the Oracle Server architect's most important trade-off constraints and give the reader some accurate information to help make wise decisions in the pursuit of his or her perfect system.

Acknowledgments

My sincerest thanks go to the following friends for their time and enlightening comments: Dominic Delmolino, Greg Doherty, Tim Gorman, Todd Guay, Deepak Gupta, Gary Hallmark, Andrew Holdsworth, Phil Joel, Anjo Kolk, Mark Pavkovic, Richard Powell, Lyn Pratt, Willis Ranney, Craig Shallahamer, Hank Tullis, Hugh Ujhazy, Peter Utzig, Mitch Wallace, and Graham Wood.

References

CHEN, P.; LEE, E.; GIBSON, G.; KATZ, R.; PATTERSON, D. 1994. "RAID: high-performance, reliable secondary

storage" in ACM Computing Surveys, Vol. 26 No. 2 (Jun 1994).

GUI, JEFFREY. 1993. "OLTP and System Reliability" in *OLTP Handbook*, edited by Gary McClain, Intertext/McGraw-Hill, New York NY.

MAULIK, B.; PATKAR, S. 1995. "Outage recovery timings" in *Technical Reports Compendium* Vol. I (Dec 1995). Oracle internal document.

MILLSAP, C. 1995a. *The OFA Standard—Oracle7 for Open Systems*. Oracle internal document, available on-line at http://www.europa.com/~orapub.

MILLSAP, C. 1995b. *Oracle7 Server Space Management*. Oracle internal document, available on-line at http://www.europa.com/~orapub.

MILLSAP, C. 1996. *Selecting the Optimal Oracle Database Block Size*. Oracle internal document. Not yet available on-line.

Oracle7 Server Administrator's Guide. 1996. Oracle standard product documentation, Redwood Shores CA.

Oracle7 Server Concepts Manual. 1996. Oracle standard product documentation, Redwood Shores CA.

PATTERSON, D.; GIBSON, G.; KATZ, R. 1988. "A case for redundant arrays of inexpensive disks (RAID)" in *International Conference on Management of Data (SIGMOD)*. ACM, New York: 109–116.

TO, L. 1995. "Outage prevention, detection, and repair" in *Technical Reports Compendium* Vol. I (Dec 1995). Oracle internal document.

Understanding Disk Arrays. 1995. Sun Microsystems white paper, Mountain View CA.

About the Author

Cary Millsap is the senior director of Oracle Services' System Performance Group. His team is responsible for building new capabilities for Oracle to deliver into its high-end technical market for large, distributed, high-performance, high-availability systems. The System Performance Group serves customers worldwide in the areas of performance management and maintenance, operational procedure design, and system architecture design.